An Extension of the Boyer-Moore Theorem Prover to Support First-Order Quantification

Matt Kaufmann[1]
Computational Logic Inc.
1717 W. Sixth St., Suite 290
Austin, TX 78703, USA
phone: (512) 322-9951 email: kaufmann@cli.com

Abstract. We describe an implementation of an extension to the Boyer-Moore Theorem Prover and logic that allows first-order quantification. The extension retains the capabilities of the Boyer-Moore system while allowing the increased flexibility in specification and proof that is provided by quantifiers. The idea is to Skolemize in an appropriate manner. We demonstrate the power of this approach by describing three successful proof-checking experiments using the system, each of which involves a theorem of set theory as translated into a first-order logic. We also demonstrate the soundness of our approach.

1. Introduction

The successful use of the Boyer-Moore Theorem Prover "NQTHM"[2] to proof-check a diversity of theorems is well-documented in [1]. Nevertheless, there are occasions when its quantifier-free logic may be somewhat awkward or inadequate because of its lack of explicit quantification. Actually, the current logic does have a version of bounded quantification, but that notion involves a somewhat tricky concept of evaluation, similar in flavor to the Lisp EVAL construct. What's more, that logic is considerably weaker than full first-order logic.[3]

We introduce here an interface from first-order logic into the Boyer-Moore logic. The main idea of our approach is to retain the current prover's strengths while allowing first-order reasoning. In fact, the examples in the final section of this paper do make use of the induction capabilities of the Boyer-Moore prover.

We should perhaps emphasize that our use of Skolemization to give the effect of quantification for a logic is standard. The interest here is that we have been able to *use* a carefully-constructed Skolemizer to proof-check some "relatively deep" theorems. By "relatively deep" we do not mean by current mathematical standards, but rather, by the standards of existing theorems whose proofs have been checked down to the minutest details by an automated reasoning program. Moreover, the implementation is stable enough that it is in general use in the community of users of the Boyer-Moore prover.

[1]This research was supported in part by ONR Contract N00014-88-C-0454. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Office of Naval Research or the U.S. Government.

[2]This awkward acronym stands for New Quantified THeoreM prover. Ask Boyer or Moore to explain this!

[3]For logicians, let us point out that all integer functions definable in the NQTHM logic are easily seen to be recursive in the halting problem, hence definable with at most two alternations of quantifiers.

The Boyer-Moore theorem prover [1] is generally considered a state-of-the-art prover for verification of algorithms. However, the lack of first-order quantification is somewhat restrictive; this work tries to remedy that situation. The papers of D. Goldschlag on verification of concurrent algorithms [2, 3, 4] and the paper of Y. Yu on group theory proof-checking [5] make use of this extension.

The rest of this introduction gives a brief description of how we add quantifiers to the Boyer-Moore paradigm.[4] In our discussion here and throughout the paper, we avoid the Lisp-like syntax of the Boyer-Moore NQTHM logic in favor of a traditional first-order syntax. More generally, our intention is to make this paper self-contained, without assuming familiarity with the Boyer-Moore logic or prover.

The main idea is to introduce a new function symbol that abbreviates a given first-order formula. Consider for example the following definition of a function named **arbitrarily-large-p**, which asserts of its input **n** that there exist arbitrarily large **y** such that **p(n,y)** holds.

```
arbitrarily-large-p(n) ≡
∀ x ∃ y (x < y ∧ p(n,y))
```

Our system Skolemizes this equivalence to obtain the following two implications.

```
[ [ x(n) < y ∧ p(n,y) ] →
  arbitrarily-large-p(n) ]

[ arbitrarily-large-p(n) →
  [ x < y(n,x) ∧ p(n,y(n,x)) ] ]
```

We hope someday to develop heuristics to support good automatic use of such implications. However, in the meantime, our interactive "proof-checker" enhancement of the prover [6], as extended to handle free variables [7], has proved quite adequate, as illustrated by the examples presented in the final section.

The following section briefly reviews the Boyer-Moore system and carefully documents how one may use quantifiers. It also includes brief discussions of our Skolemization and miniscoping (formula-simplifying, cf. deChampeaux [8]) procedures. Soundness is addressed in Section 3. We conclude in Section 4 with brief descriptions of three successful experiments employing the new quantifiers mechanism. The full input files for those proofs may be found in [9], where one may also find further details and proofs of correctness of our particular brands of Skolemization and miniscoping.

Let us emphasize that the main interest here is in the nontrivial examples that have been successfully completed with a working implementation. These examples are summarized in the final section. The following sections serve simply as documentation for the enhanced logic and system.

Acknowledgements. I'd like to thank my colleagues at Computational Logic for useful conversations and suggestions during the course of this work. I especially thank Bob Boyer and J Moore for useful suggestions, as well as the referees of this and of a previous version of the paper.

[4]Motivating remark for logicians: the result is essentially first-order Peano Arithmetic or, equivalently, first-order finite set theory.

2. Documentation

The Boyer-Moore Theorem Prover and its logic are carefully and thoroughly documented in [1]. The logic is a variant of Pure Lisp [10], which in turn may be viewed as an equational variant of first-order logic. There is an inference rule for induction and a rule for recursive definition. A session with the system may include definitions, axioms, and theorems to be proved. Theorems may be annotated to be stored as rewrite rules, and such theorems are used automatically by the system in future proofs. User actions such as definitions and theorems are sometimes referred to as *events*. A sequence of events built on top of the "ground-zero" (built-in) logic is sometimes called a *history*.

This note introduces a new NQTHM event, DEFN-SK, together with a corresponding extension to the Boyer-Moore logic. The "SK" is in honor of Thoralf Skolem, for whom the various well-known Skolemization algorithms are named. Skolemization maps a first-order formula to a formula without quantifiers, and hence gives us a way to "interpret" first-order formulas in a slight extension of the Boyer-Moore logic, with minimal changes to the theorem prover.

Although the Boyer-Moore Theorem Prover is fully automatic in one sense, it is quite interactive in a more fundamental sense. Typically the user submits a theorem, look at the prover's output to see where the proof failed, and then proves additional rewrite rules before trying the proof again, until finally the proof is obtained "automatically".[5] In fact it is quite common for users to undo definitions, fixing "bugs" in them before trying proofs again that previously failed. Hence, we want the output of our Skolemization procedure to provide terms that are helpful, i.e., both "strong" and somehow intuitive, when we attempt to reason about them. We have therefore decided to provide a rather basic Skolemization procedure that doesn't provide too many surprises, but to allow a mildly "smart" *miniscoping* process. The input formula may then be replaced by a "better" formula that is equivalent (by *miniscoping*) to it; this "better" formula is the one that is Skolemized.

The syntax for formulas is given in Subsection 2.1. Skolemization and miniscoping are discussed in Subsections 2.2 and 2.3 respectively. The DEFN-SK event is documented in Subsection 2.4. We close in Subsection 2.5 with some remarks.

2.1. Formulas

Fix a history, i.e. a sequence of events (including definitions and theorems). A Boyer-Moore *term* is (as specified in [1]) a variable symbol or a function application of the form $\mathtt{f(t_1, \ldots, t_n)}$, where $\mathtt{f}$ is defined in the current history (or is built-in) and each $\mathtt{t_i}$ is a term. A (first-order) *formula* is either a Boyer-Moore term, a *propositional combination of* formulas, or a *quantification of* a formula. By a *propositional combination of* formulas we mean any expression of the form $(\neg\ \mathtt{x})$, $(\mathtt{x} \wedge \mathtt{y})$, $(\mathtt{x} \vee \mathtt{y})$, $(\mathtt{x} \rightarrow \mathtt{y})$, or $(\mathtt{IF\ t\ THEN\ x\ ELSE\ y})$, where $\mathtt{x}$ and $\mathtt{y}$ are formulas and $\mathtt{t}$ is a term. A *quantification of* a formula is any expression of the form $(\forall\ \mathtt{v})\ \mathtt{x}$ or $(\exists\ \mathtt{v})\ \mathtt{x}$, where $\mathtt{v}$ is a variable and $\mathtt{x}$ is a formula. As usual we may write $(\mathtt{x_1} \wedge \mathtt{x_2} \wedge \ldots \wedge \mathtt{x_n})$ as an abbreviation for $(\mathtt{x_1} \wedge (\mathtt{x_2} \wedge (\ldots (\mathtt{x_{n-1}} \wedge$

[5]Some users get more direct control of the proof process by using the PC-NQTHM interactive enhancement [6].

$x_n) \ldots)))$; similarly for '$\vee$'. We also allow $(Q\ v_1\ \ldots\ v_n)\ x$, where Q is $\forall$ or $\exists$ and n is a non-negative integer, as an abbreviation for $(Q\ v_1\ (Q\ v_2\ \ldots\ (Q\ v_n\ x)\ \ldots))$. Finally, $(x \Leftrightarrow y)$ abbreviates $((x \rightarrow y) \wedge (y \rightarrow x))$

We will feel free to use familiar notions about formulas, such as the notion of *free variable*. Such notions are explained in any logic textbook; see for example [11].

2.2. Skolemization in Brief

For now let us discuss Skolemization in an abstract framework. In fact our actual algorithm is quite standard so we omit it here (see [9] for details), although we will comment on it below.

Let *F* be a set of function symbols and let ϕ be a first-order sentence (i.e. formula without free variables), all of whose function symbols are contained in *F*. Let ψ be a quantifier-free formula (i.e. term) with universal closure[6] ψ'. Then ψ is said to be a *Skolemization of* ϕ *with respect to F* if for any set *S* of first-order sentences, all of whose function symbols are included in *F*, (1) the extension of *S* by the sentence $[\phi \rightarrow \psi']$ is a conservative extension of *S*,[7] and (2) the sentence $[\psi' \rightarrow \phi]$ is a theorem of *S*. If we don't mention *F*, then it is taken to be the union of the set of function symbols in ϕ together with those of the current history. The function symbols of ψ that do not occur in *F* are called the *Skolem functions* of ψ.

Consider for example the formula $(\forall\ x)\ (\exists\ y)\ p(x,y)$. It's easy to see that $p(x, f(x))$ is a Skolemization of this formula in the sense above, as long as f does not occur in that formula or in the current history. However, on more complicated formulas one can get different results depending on the particular Skolemization algorithm used. Briefly, our particular algorithm Skolemizes from the outside in, and Skolem functions depend only on the variables in their scopes. For example, consider the following definition of a formula ϕ:

$$\phi \Leftrightarrow \exists\ x\ (p(x) \rightarrow \forall\ y\ p(y))$$

Our Skolemization of this biconditional (*not* just of ϕ!) is as follows, where **sk-x** and **sk-y** are the Skolem functions introduced, and hence **sk-x()** and **sk-y()** are the applications of these functions to no arguments, i.e. they are constants.

$$\begin{array}{l}
[\ [\ p(x) \rightarrow p(\text{sk-y}())\ ] \\
\quad \rightarrow \phi\ ] \\
\wedge \\
[\ \neg\ [\ p(\text{sk-x}()) \rightarrow (p\ y)\ ] \\
\quad \rightarrow \neg\ \phi\ ]
\end{array}$$

[6]Recall that the *universal closure* of a formula ψ is a formula $(\forall\ v_1\ \ldots\ v_n)\ \psi$ where $(v_1\ \ldots\ v_n)$ is an enumeration of the free variables of ψ.

[7]That is, it proves no new theorems in the language of *S*.

Thus, to prove ϕ, it suffices to find an **x** such that we can prove

```
p(x) → p(sk-y())
```

and of course, the application of the function **sk-y** to no arguments serves as the desired **x**. Compare this approach with the approach of letting the Skolem functions depend on "governing" quantified variables that have been encountered as one dives into the formula. With the latter approach, the Skolem function **sk-y** will have **x** as an argument, and we will may have trouble proving ϕ since no instantiation for **x** turns the term **p(x) → p(sk-y(x))** into a tautology. (As Bob Boyer points out, even in the latter approach a resolution proof is found quite trivially; but our point here is to treat Skolemization in a manner that is *convenient* in the context of the Boyer-Moore rewriting approach.)

2.3. Miniscoping in brief

We have just seen the desirability of reducing the number of variables on which the Skolem functions depend. One finds a method in deChampeaux [8] that has such an effect; the idea, which is sometimes called *miniscoping*, is to push quantifiers inward as far as possible.[8] In this work we choose a very simple method along these lines. Consider the following example.

```
q(z)  ⇔  ∃ x ∀ y (p(x,z) → p(y,z))
```

A natural Skolemization provides Skolem function for **y** that depends on both **x** and **z**:

```
[ [ p(x,z) → p(sk-y(x,z),z) ]
  → q(z) ]
∧
[ ¬ [ p(sk-x(z),z) → p(y,z) ]
  → ¬ q(z) ]
```

As in the previous example, the first conjunct is not quite sufficient to prove **q(z)** "naturally". But if we first move the **(∀ y)** quantifier inward (which is legal since **y** does not occur in **p(x, z)**), and then move **(∃ x)** inward, we obtain

```
q(z) ≡ [ ((∀ x) p(x,z)) → ((∀ y) p(y,z)) ]
```

and this has a "better" Skolemization:

[8]However, a referee has pointed out that there are often occasions where distribution of existential quantifiers over disjunctions can result in introduction of more Skolem functions than is desirable.

```
[ [ p(x,z) → p(sk-y(z),z) ]
   → q(z) ]
∧
[ ¬ [ p(sk-x(z),z) → p(y,z) ]
   → ¬ q(z) ]
```

For, now **q(z)** is clear by instantiating **x** to **sk-y(z)** in the first conjunct above.

All we require of this *miniscoping* procedure is that it transform every first-order formula to one that is logically equivalent to it.

2.4. New event: Defining first-order notions

We extend the Boyer-Moore logic by adding a new "axiomatic act" called DEFN-SK. The following "formal section" describes the extended logic, while the "manual section" describes our implementation. Let us fix Skolemization and miniscoping algorithms meeting the specifications given in the preceding subsections.

(formal section)

Let ϕ be a first-order formula with respect to a given history (sequence of axioms) *h* whose free variables are contained among the set $\{v_1, \ldots, v_n\}$, where $v_1, \ldots, v_n$ are distinct variables. Let **g** be a function symbol of arity **n** that does not occur in *h*. Suppose that **sk** is a Skolemization of the first-order sentence

$$(\forall\ v_1\ \ldots\ v_n)\ (g(v_1,\ldots,v_n) \Leftrightarrow \phi)$$

with respect to a set of function symbols that contains **g** together with all function symbols of *h*. Then it is permitted to extend *h* by adding **sk** as an axiom.

(manual section)

General Form: (DEFN-SK name args form &optional directives)
Example Form (in traditional first-order syntax):

```
DEFN-SK:
bar(z) ⇔ (∃ x) (∀ y) [p(x,z) → p(y,z)]
FORMULA:   ((∀ x) p(x,z)) → ((∀ y) p(y,z))
PREFIX:  sk-
```

Here

- **name** is the name of the new Boolean-valued function representing the (quantified) notion being introduced, e.g. **bar** in the example above.
- **args** is the list of formal parameters for this new function. In the example above, the only formal parameter is **z**.
- **form** is a first-order formula (in the "extended syntax" described above), e.g. **(∃ x) (∀ y) (p(x,z) → p(y,z))** in the example above.
- **directives** is a list of instructions on how to store this event. For example, the "**FORMULA** directive", which is **((∀ x) p(x,z)) →**

((∀ y) p(y,z)) in the example above, is a first-order formula such that the result of applying the miniscoping procedure to that formula and to **form** must be the same, up to renaming of bound variables. In that case, Skolemization is applied to the formula supplied there rather than to the original formula. The "**PREFIX** directive" in the example above, "**PREFIX: sk-**", indicates that each Skolem function name will begin with the string "**SK-**". The directives, which can also include one of the form "**SUFFIX: ...**", are optional.

Let us comment further on this form. First, **args** should be a list of distinct variables that includes all free variables of **form**. Let **args** be **(arg_1 ... arg_n)**, let **u** be **name(arg_1,...,arg_n)**, and let **body** be **form** unless a **FORMULA** directive is supplied, in which case **body** is that formula. Consider the following first-order sentence:

```
(*)    (∀ arg₁ arg₂ ... argₙ) (u ⇔ body) .
```

The effect of this event is to add as an axiom a Skolemization of this sentence, with respect to the result of adding **name** to the set of function symbols of the current history. In fact this axiom is added as a pair of rewrite rules (as described in the next subsection). Moreover, the entire axiom becomes the formula associated with **name**[9], so that it may be referred to in, for example, a USE hint of a PROVE-LEMMA form. If a **PREFIX** or **SUFFIX** directive is supplied, then the indicated prefix (and/or suffix) is tacked on to the front (respectively, back) of the name of each Skolem function generated. Finally, the system stores the fact that **name** always returns **T** or **F**.

2.5. Further remarks

(i) A macro DEFN-FO is provided for generating good **FORMULA** directives. DEFN-FO works by applying miniscoping to the given formula and, provided that the result differs from the given formula, generating a **FORMULA** directive with that result.

(ii) Consider the example above. In that case, the formula associated with **bar** is the conjunction of the following two formulas, each of which is made into a rewrite rule (also) named BAR. Notice that the second of these is written as the contrapositive of what one might have otherwise expected, in order to put it in a form guaranteed appropriate for a rewrite rule. The conclusion of the first is handled just as though it were **(bar(z) = T)** (where **T** is "true"), while the conclusion of the second is handled just as though it were **bar(z) = F**.

```
[ [ p(x,z) → p(sk-y(z),z) ]
  → bar(z) ]

[ ¬ [ p(sk-x(z),z) → p(y,z) ]
  → ¬ bar(z) ]
```

(iii) We claim that it is a theorem of the new history that the application of **name**

[9]for Boyer-Moore experts: the result of evaluating **(FORMULA-OF 'name)**

to **args** is equivalent to **form**. That is, the formula (*) in the "formal section" above is a theorem of the new history. This is because the new axiom provably yields (*), by the specification of Skolemization given in Subsection 2.2 above. Conversely, it is conservative to add this axiom, in a sense we now make precise.

3. Soundness

In this section we state and prove theorems that demonstrate the soundness of our approach. It is helpful to recall that DEFN-SK extends a history by adding a *term* in the (quantifier-free) logic. However, it was shown above that the first-order formula (*) implicit in that DEFN-SK event is indeed a theorem of the resulting history (when that history is viewed as a set of first-order formulas).

First, recall the following standard definition: a set S_2 of first-order sentences that contains a set S_1 is a *conservative extension* of S_1 if any theorem provable from S_2 that is in the language (alphabet) of S_1 is in fact already a theorem of S_1.

The following simple theorem shows that DEFN-SK gives conservative extensions. It follows that consistency is preserved by DEFN-SK, since conservative extensions preserve consistency: if S_2 is a conservative extension of a consistent set S_1 then since **F** (*false*) is not a theorem of S_1, **F** is not a theorem of S_2.

Soundness Theorem for DEFN-SK[10]. Suppose we embed the Boyer-Moore logic into a traditional first-order logic, such as that of [11], turning the induction principle into a collection of axioms, admitting existential quantifiers and the existential-quantifier introduction-rule. Then a DEFN-SK event, as defined above, results in a conservative extension of the previous theory.

Proof. Suppose we extend a history h to a new history h' by adding a Skolemization of the first-order sentence **(∀ v1 ... vn) (g(v1,...,vn) ⇔ body)**, where all free variables of **body** are among **{v1, ..., vn}** and **g** is a new function symbol (and the Skolemization is with respect to some set containing **g** along with all function symbols of h). Let h_1 be the history obtained by adding the above first-order sentence to h, and let h_2 be the union of h_1 and h'. Then h_1 is a conservative extension of h since it's simply a definitional extension (cf. [11]). And h_2 is a conservative extension of h_1 by the property of Skolemization given in Subsection 2.2. So for any first-order sentence **A** in the language of h, if $h' \vdash$ **A** then $h_2 \vdash$ **A** (since h_2 extends h'), hence $h_1 \vdash$ **A** (by conservativity of h_2 over h_1), hence $h \vdash$ **A** (by conservativity of h_1 over h). ⊣

The paper [12] contains an argument that shows the correctness of the implementation of an event called FUNCTIONALLY-INSTANTIATE, which, roughly speaking, allows one to replace some function symbols in a theorem with others that satisfy those symbols' defining axioms. However, that argument assumes an underlying logic that does not have DEFN-SK. In Appendix C of [9] we provide the main lemma required to extend the arguments in [12] to the case that FUNCTIONALLY-INSTANTIATE is added to a version of the logic that includes

[10]This is analogous (even to its wording!) to a corresponding note about the new CONSTRAIN event in [12].

DEFN-SK.

4. Examples

The purpose of this section, indeed of this paper, is to demonstrate that our DEFN-SK interface from first-order logic into the Boyer-Moore logic enables one to mechanically proof-check interesting theorems. We treat three separate examples here. Complete proof scripts may be found in the final three appendices of [9]. Our intention in this section is to give just enough detail so that the reader can see what it is that we have verified and to get some idea of our approach.

All three examples are translations of theorems of set theory into our first-order extension of Boyer-Moore logic. While we do not claim that all of set theory have such natural translations, still we feel that these examples demonstrate that our approach can deal with interesting set-theoretic results.

All of these examples introduce axioms using the CONSTRAIN mechanism reported in [12]. CONSTRAIN is simply a consistent (in fact, conservative) way of adding axioms. The first subsection below says a little more about CONSTRAIN; full details may be found in [12].

All three examples below make heavy use of the "proof-checker" enhancement (PC-NQTHM) of the Boyer-Moore prover, as reported first in [6] and then extended in [7] to implement a notion of free variables. The proofs especially used the *macro command* SK* documented in [7] to eliminate notions defined by DEFN-SK. Space does not permit a detailed description here of macro commands in general or SK* in particular; suffice it to say that SK* works by automatically applying the Skolem axioms and backchaining. In fact, in the first two examples we were able to extract, usually without much difficulty, the requisite applications of the Skolem axioms so that we could replace "low-level" use of PC-NQTHM by hints to the NQTHM prover. We hope to implement something akin to the proof-checker's treatment of free variables, as used in these proofs, in the Boyer-Moore prover someday; then the aforementioned USE hints should not, we hope, be necessary.

We continue to use familiar first-order syntax in place of the official Boyer-Moore Lisp-like syntax, which may be found in [9]. Comments are to be found in italics, preceded by semicolons.

4.1. Koenig's Tree Lemma.

Koenig's Tree Lemma states that every infinite, rooted, finitely-branching tree has an infinite branch. Rather than explain these terms here, we will simply state the axiom in our formalization of Koenig's Tree Lemma after providing some intuition.

We identify a *node* of such a tree with the sequence of positive integers that represents the path from the root of the tree to that node, most recent "turn" first. For example the node `x321` pictured below corresponds to the sequence `(3 2 1)`.

```
r121  r221 r321    r112
    \   |  /      /
r11   r21      r12
  \  /         /
   r1        r2
     \      /
        r
```

In particular, the empty sequence **nil** represents the root of the tree. And the less-than relation is represented by the notion of "terminal subsequence", which (in the usual style of the Boyer-Moore logic) we define recursively.

In the Boyer-Moore text below, comments appear in *italics* and are indicated by placing a semicolon to indicate that the rest of that line is a comment. Here we are making a definition in the Boyer-Moore logic (albeit using familiar first-order syntax rather than Lisp syntax), according to a principle of (possibly recursive) definition that is conservative when the Boyer-Moore logic is viewed as a subset of first-order logic with induction. So for example, the following definition defines **subseq** to be a function of two arguments, called **s1** and **s2**, which returns the indicated value. The predicate **listp** holds of *non-empty* sequences, while the function **cdr** takes the tail of a list.

```
DEFINITION.
subseq(s1,s2) ≡
;; s1 is the result of popping some elements off the top of s2
(IF (s1 = s2)
    THEN t ;; true
    ELSE
    (IF ¬ listp(s2)  ;; if s2 is not a pair, then false:
        THEN f
        ELSE
        ;; otherwise, s1 and s2 are distinct and s2 is not empty,
        ;; so we check that s1 is a subseq of the tail of s2.
        subseq(s1,cdr(s2))))
```

Here then is the axiom that introduces an appropriate tree, by way of a predicate **node-p** that recognizes the nodes of (i.e. finite paths through) the tree. That is, node-p recognizes the legal paths. We enforce that the tree is finitely-branching by introducing a function **succard** ("Successors Cardinality") that, for a given node in the tree, returns the number of immediate successors of that node. The CONSTRAIN mechanism guarantees consistency of the axiom by requiring us to give example functions that make the axiom true. In fact, the theorem prover checks the conservativity (hence consistency) of the following axiom by checking that it holds when **node-p** is replaced by **all-ones**, **succard(s)** is replaced by **1** (for all **s**), and **s-n(n)** is replaced by **ones(n)** (for all **n**). As usual, we take minor liberties with the syntax, for readability; the precise event form may be found in [9]. The function **cons** is the list constructor; thus, **cons(a,s)** represents the result of tacking **a** on to the front of the list **s**.

```
CONSTRAIN event (constraining axiom)
NAME of event:  KOENIG-INTRO

AXIOM:

node-p(nil)  ;; nil is the root
```

```
∧
;; node-p is a predicate
[ node-p(s) = t ∨ node-p(s) = f ]
∧
;; the successors of s are determined by succard (Successors Cardinality)
[ node-p(s) →
    [node-p(cons(n,s)) ⇔
     ;; n is in the set {1,  ...,  succard(s)}
     (0 < n  ∧  n ≤ succard(s))] ]
∧
;; The tree is closed under initial subsequences.
[ (node-p(s1) ∧ subseq(s,s1)) → node-p(s) ]
∧
;; We stipulate that the tree is infinite by saying that s-n is a one-one enumeration of nodes.
node-p(s-n(n)) ∧ [(0 ≤ i ∧ 0 ≤ j ∧ i ≠ j) → s-n(i) ≠ s-n(j)]
∧
;; Nodes are proper lists, i.e. lists terminating in nil – a technicality we won't explain further.
[ ¬ plistp(s) → ¬ node-p(s) ]

WITNESSES:
LET
 node-p(s) ≡ all-ones(s)   ;; s consists of all ones
 succard(s) ≡ 1
 s-n(n) ≡ ones(s)   ;; sequence of n ones

HINT to theorem prover for proof that the witnesses satisfy the axiom:
(DISABLE subseq)
```

Our approach is to follow the natural proof in which one builds a branch through the tree "nonconstructively" as follows. Starting at the root, we maintain the invariant that the top node on the branch constructed so far has infinitely many successors (**not** *immediate* successors, of course!) in the tree. So given the current top node, we extend the branch by choosing an immediate successor that has infinitely many successors -- note that if each immediate successor had only finitely many successors, then the given top node would have only finitely many successors (since the tree is finitely-branching), which would violate the invariant.[11] Here is the DEFN-SK event that formalizes the notion of "**s** has infinitely many successors", in the sense that **s** has arbitrarily high successors in the tree.

```
DEFN-SK:
inf(s)  ⇔
(∀ big-h) (∃ big-s)
 [subseq(s,big-s) ∧ node-p(big-s) ∧ big-h < length(big-s)]
```

This adds the Skolem axiom

[11]Comment for logicians. It's well known that there are infinite, recursive, finitely-branching, rooted trees without infinite recursive branches, and in fact this result relativizes to any oracle. This strongly suggests that there is no way even to formulate Koenig's Lemma in the unmodified, constructive Boyer-Moore logic.

```
[ [ subseq(s,big-s) ∧
    node-p(big-s) ∧
    big-h(s) < length(big-s) ]
  → inf(s) ]
∧
[ ¬ [ subseq(s,big-s(big-h,s)) ∧
      node-p(big-s(big-h,s)) ∧
      big-h < length(big-s(big-h,s)) ]
  → ¬ inf(s) ]
```

Our method for reasoning about **INF** is to use the PC-NQTHM [6] enhancement of the Boyer-Moore prover, as extended to support free variables [7], to deal with the free variables introduced by backchaining with the Skolem axioms. (See the brief discussion on SK* preceding this subsection.) The final theorem is as follows. It says that the function **k** enumerates an infinite sequence of nodes, each of which is an initial subsequence of the next, such that the nth node has height n for all n.

```
n ≥ 0
→
[ node-p(k(n)) ∧
  ( j ≥ i → subseq(k(i),k(j)) ) ∧
  length(k(n)) = n ]
```

Let us make one last technical point. A key to the proof is the following definition, which is explained below.

```
DEFINITION.
all-big-h(s,n) ≡
(IF n=0
    THEN (length(s) + 1) ;; then return the successor of the length of s
    ;; otherwise, return a "big enough" number, as explained below
    ELSE
    big-h(cons(n,s)) + all-big-h(s,n - 1))
```

This function returns a number that is bigger than **big-h(cons(s,i))** for all positive **i** ≤ **n** and is also bigger than length of **s**. Here, **big-h** comes from the definition (DEFN-SK) of **inf** above; roughly, **big-h** is a function that has the property that if there is a node of length at least **big-h(s)** extending a given node **s**, then there are nodes of arbitrarily large height extending **s**. When we are looking to extend a given node **s** to a branch as in the proof outline above, the idea is to use the **inf(s)** hypothesis (invariant) to find an extension **s1** of **s** of length at least **all-big-h(s,succard(s))**. We may then note that **s1** extends some immediate successor **s0** of **s**, and by definition of **all-big-h** this node is high enough in the tree to guarantee **inf(s0)**.

4.2. Ramsey's Theorem.

Ramsey's Theorem for exponent 2 says the following (in the finite version). Let **P** be a partition of the 2-element subsets of a given infinite set **A** into a finite number of pieces, say **n** pieces. (One can visualize here a space of nodes where each pair of nodes is connected by an edge of one of **n** given colors.) Then there is an infinite subset **H** of **A** such that all pairs from **H** lie in the same piece of the partition. (In terms of our graph picture, all edges between pairs of nodes from **H** are of the same color.)

This theorem has a proof that is rather similar in flavor to the Koenig's lemma proof described in the immediately preceding subsection.[12] For simplicity we assume that the partition is on pairs of natural numbers, and we call it **p-num**. However, we immediately define a function **p** that makes sense for all pairs (by coercing them to numbers for **p-num**). Think of **bound()** below as the number of "colors"; we are partitioning the pairs into the set {1, ..., **bound()** }.

```
CONSTRAIN event (constraining axiom)
NAME of event:  P-NUM-INTRO

AXIOM:

[ 0 ≤ x ∧ 0 ≤ y ]
→
[ 0 < p-num(x,y) ∧
  bound() ≥ p-num(x,y) ∧
  p-num(x,y) = p-num(y,x) ]

WITNESSES:
LET
 p-num(x,y) ≡ 1
 bound() = 2

DEFINITION ;; a technicality, needed since the logic is untyped
p(x,y) ≡
;; Coerce x and y to natural numbers before applying p-num to them.
p-num(fix(x),fix(y))
```

We follow along a standard proof of this theorem. One defines a notion of *prehomogeneous set*, which says that for numbers **i < j** in such a set, the value of the partition on the pair **{i,j}** should depend only on **i**. In order to formalize this notion we actually consider lists of pairs **<i,c>** whose first component **i** is the number and whose second component **c** is the intended color. The auxiliary function **PREHOM-SEQ-1** takes arguments **a** and **x**, where one may think of **x** as being a list of pairs **<i,c>** as above, each **i** being less than **a**, where this function checks that **c** is the right color for the pair **{i,a}**. In the following definition, we take some syntactic liberties in order to increase readability, using pattern-matching where none really exists in the NQTHM logic, and using notation **[...]** to represent lists.

[12] A proof of a formalization of this version of Ramsey's Theorem has also been carried out by Ketonen [13].

```
DEFINITION.
;; x is a list of pairs <i,c>, and this says that for each such pair, p(i,a) = c.
prehom-seq-1(a, []) ≡ t ;; return true for the empty list

prehom-seq-1(a, [<i_1,c_1>, <i_2,c_2>, ..., <i_n,c_n>]) ≡
[ p(i_1,a) = c_1 ∧
  prehom-seq-1(a, [<i_2,c_2>, ..., <i_n,c_n>]) ]
```

```
DEFINITION.
;; Recognizes lists of pairs such that if <j,c'> precedes <i,c>, then i < j and p(i,j)=c.
prehom-seq ([]) ≡ t ;; The empty list is prehomogeneous.

prehom-seq ([<i,c>]) ≡ t ;; A singleton list is prehomogenous.

prehom-seq ([<i_1,c_1>, <i_2,c_2>, ..., <i_n,c_n>]) ≡
[ i_2 < i_1 ∧
  prehom-seq-1(i_1, [<i_2,c_2>, ..., <i_n,c_n>]) ∧
  prehom-seq([<i_2,c_2>, ..., <i_n,c_n>]) ]
```

Continuing with the proof: the idea now is to define an increasing sequence of natural numbers that forms a prehomogeneous sequence. Each member **i** of this sequence gets a color such that the value of the partition on any pair **{i,j}**, where **j** is also in the sequence and **i < j**, depends only on the color associated with **i**. But since there are only finitely many colors, one must appear infinitely often in this prehomogenous sequence. Then the subsequence corresponding to that color forms the desired homogeneous set.

However, in order to construct the desired prehomogeneous sequence we need a stronger invariant than prehomogeneity. That invariant is expressed by the first of two introductions of quantifiers in this proof, which says roughly that **s** has arbitrarily large extensions to a prehomogeneous sequence:

```
DEFN-SK:
extensible (s) ⇔
;; holds iff there are infinitely many a such that prehom-seq-1(a,s).
(∀ above) (∃ next)
  [ above < next ∧ prehom-seq-1(next,s) ]
```

We omit here the definition of **ramsey-seq(n)**, which is a strictly decreasing prehomogeneous sequence of length **n**, and of **ramsey-index(n)**, which uses auxiliary quantifier definitions to return the index of the **n**th member of **ramsey-seq** that has the "right" color. Here is our definition of the desired prehomogeneous sequence, where the function **car** both takes the first member of a list and takes the first component of a pair.

```
DEFINITION.
ramsey(n) ≡
  car(car(ramsey-seq(ramsey-index(n))))
```

The final theorems are as follows. They say that the sequence of numbers **ramsey(i)**, as **i** runs through the positive natural numbers, is a strictly increasing sequence that is homogeneous. We omit the definition of the constant (function) **color** here, since how it is defined is not particularly important, as long as the following theorems can be proved.

```
i < j → ramsey(i) < ramsey(j)

(0 ≤ i ∧ 0 ≤ j ∧ i ≠ j) →
p-num(ramsey(i),ramsey(j)) = color()
```

4.3. Schroeder-Bernstein Theorem.

The Schroeder-Bernstein Theorem says that for any sets **a** and **b**, if there is a one-to-one function from **a** to **b** and also a one-to-one function from **b** to **a**, then there is a bijection from **a** onto **b**.[13] We followed the proof sketch given in Exercise 8 of Chapter 1 of [14]. The following axiom introduces our assumptions.

```
CONSTRAIN event (constraining axiom)
NAME of event:  FA-AND-FB-ARE-ONE-ONE

AXIOM:

;; fa is one-to-one
[ (a(x) ∧ a(y) ∧ x ≠ y) →
  fa(x) ≠ fa(y) ]
∧
;; fb is one-to-one
[ (b(x) ∧ b(y) ∧ x ≠ y) →
  fb(x) ≠ fb(y) ]
∧
;; the image of fa on a is contained in b
[ a(x) → b(fa(x)) ]
∧
;; the image of fb on b is contained in a
[ b(x) → a(fb(x)) ]
;; the characteristic functions a and b are boolean-valued
∧
(a(x) = t ∨ a(x) = f) ∧ (b(x) = t ∨ b(x) = f)

WITNESSES:
LET
 fa and fb be the identity function
 a(x) ≡ t
 b(x) ≡ t
```

Here is the idea of the proof. Imagine that sets **a** and **b** and functions **fa** and **fb** which satisfy the CONSTRAINt above. Now imagine that we take the images of **a** and **b** under **fa** and **fb**, respectively. Let us write **a0** and **b0** to denote **a** and **b**, respectively, and let **a1** and **b1** be the respective images. We can of course repeat this process as

[13]Others have mechanically verified versions of this theorem, in particular, a Theory of Constructions proof (perhaps first done by Gerard Huet) and one by a system under development by Kurt Ammon (personal communication).

many times as we like, letting `a(i+1)` be the image of `b(i)` under `fb` and `b(i+1)` be the image of `a(i)` under `fa`. Let `a*` be the intersection of the sets `a(i)` and let `b*` be the intersection of the sets `b(i)`.

```
                           --> fa ---->
     --------------------              --------------------
     |                  |              |                  |
     |  --------------  |              |  --------------  |
     |  |  ----------  |  |            |  |  ----------  |  |
     |  |  |  ____  |  |  |            |  |  |  ____  |  |  |
     |  |  |  \_a*_/  |  |  |          |  |  |  \_b*_/  |  |  |
     |  |  |    ..    |  |  |          |  |  |    ..    |  |  |
     |  |  --- a2 ---  |  |            |  |  --- b2 ---  |  |
     |  ----- a1 -----  |              |  ----- b1 -----  |
     |                  |              |                  |
     -------- a0 --------              -------- b0 --------
                           <---- fb <--
```

We say that a point in `a(i)` is *circled by* `i`. For $x \in a$ with $x \notin a^*$, let us define the *a-level of* `x` *in* `a` to be that `i` such that `x` belongs to `a(i)` but not to `a(i+1)`, i.e. that `i` such that `x` is circled by `i` but not by `i+1`. Define analogous notions for `b`. Then a one-to-one correspondence may be constructed by mapping the even `a`-levels and `a*` into `b` *via* `fa` and by mapping the odd `a`-levels into `b` *via* the inverse of `fb`.

We give only a brief proof outline here. Below are a few of the key definitions and lemmas to give some flavor of the proof, but we do *not* claim that this list is in any way complete. A complete input file may be found in [9]. The reader unfamiliar with Lisp should read `'a` and `'b` below as any distinct constants.

```
DEFN-SK:
in-fa-range(x) ⇔
  (∃ fa-1) ( a(fa-1) ∧ fa(fa-1) = x )
;; Similar definition of in-fb-range is omitted here.

DEFINITION.
circled(flg,x,n) ≡
;; If flg is 'a, returns t iff x is in a-n in the sense of Kunen's proof.  Similarly for b if flg is not 'a.
(IF (flg = 'a)
    THEN
    (IF n=0 THEN a(x)
        ELSE ( in-fb-range(x) ∧ circled('b,fb-1(x),n - 1) ))
    ELSE
    (IF n=0 THEN b(x)
        ELSE ( in-fa-range(x) ∧ circled('a,fa-1(x),n - 1) )))

DEFN-SK
a-core(x) ⇔
;; Introduced "inductively" so that the level will be tight -- similar definition of b-core is omitted here.
[ a(x) ∧
  (∀ a-level) [ ( 0 ≤ a-level ∧ circled('a, x, a-level) )
                      → circled('a, x, a-level + 1) ] ]

LEMMA.  A-CORE-NECC ;; similar lemma B-CORE-NECC is omitted
¬ circled('a,x,n)  →  ¬ a-core(x)
```

```
LEMMA.  A-CORE-SUFF ;; similar lemma B-CORE-SUFF is omitted
[ ( a(x) ∧ [ ( 0 ≤ a-level(x) ∧ circled('a, x, a-level(x)) )
                  → circled('a, x, a-level(x) + 1) ] )
  → a-core(x) ]

DEFINITION.
parity(n) ≡
(IF n=0 THEN t
    ELSE ¬ parity(n - 1))

DEFINITION.  ;; the isomorphism
j (x) ≡
(IF ( a-core(x) ∨ parity(a-level(x)) )
    THEN fa(x)
    ELSE fb-1(x))

DEFINITION.  ;; the isomorphism's inverse
j-1(y) ≡
(IF ( b-core(y) ∨ ¬ parity(b-level(y)) )
    THEN fa-1(y)
    ELSE fb(y))

LEMMA.  B-CORE-FA
;; fa maps a-core, and only a-core, into b-core -- similar lemma A-CORE-FB is omitted here
a(x) → ( b-core(fa(x)) ⇔ a-core(x) )

LEMMA.  CIRCLED-MONOTONE
( circled(flg,x,j) ∧ j ≥ i )
→ circled(flg,x,i)

LEMMA.  B-LEVEL-FA
;;fa increases the level by 1; similar lemma A-LEVEL-FB omitted here
( a(x) ∧ ¬ a-core(x) )
→ b-level(fa(x))  =  a-level(x) + 1

LEMMA.  J-1-J
a(x) → j-1(j(x)) = x

LEMMA.  A-CORE-FA-1
in-fa-range(y) →
[ a-core(fa-1(y)) ⇔ b-core(y) ]

LEMMA.  A-LEVEL-FA-1
[ (b y) ∧ ¬ b-core(y) ∧ in-fa-range(y) ]
→ a-level(fa-1(y)) =  b-level(y) - 1

LEMMA.  J-J-1
b(y) → j(j-1(y)) = y

LEMMA.  J-IS-ONE-ONE
( a(x1) ∧ a(x2) ∧ x1 ≠ x2 )
→  j(x1) ≠ j(x2)
```

```
DEFN-SK
j-iso() ⇔
[  ;; j maps a into b
   (∀ x) ( a(x) → b(j(x)) ) ∧
  ;; j is one-one
   (∀ x1 x2)
   [ ( a(x1) ∧ a(x2) ∧ j(x1) = j(x2) )
     →  x1 = x2 ] ∧
  ;; j is onto
   (∀ y) ( b(y) → (∃ x) ( a(x) ∧ j(x) = y ) )

THEOREM.  J-IS-AN-ISOMORPHISM
  j-iso()
```

References

1. R. S. Boyer and J S. Moore, *A Computational Logic Handbook,* Academic Press, Boston, 1988.

2. David M. Goldschlag, "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover", *IEEE Transactions on Software Engineering,* Vol. SE-16, No. 9, September 1990.

3. David M. Goldschlag, "Proving Proof Rules: A Proof System for Concurrent Programs", *Compass '90,* June 1990.

4. David M. Goldschlag, "Mechanizing Unity", in *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods,* M. Broy and C. B. Jones, eds., Elsevier Science Publishers B.V., 1990.

5. Yuan Yu, "Computer Proofs in Group Theory", *J. Automated Reasoning,* Vol. 6, No. 3, September 1990.

6. Matt Kaufmann, "A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover", Tech. report 19, Computational Logic, Inc., May 1988.

7. Matt Kaufmann, "Addition of Free Variables to an Interactive Enhancement of the Boyer-Moore Theorem Prover", Tech. report 42, Computational Logic, Inc., May 1989.

8. D. de Champeaux, "Subproblem Finder and Instance Checker, Two Cooperating Modules for Theorem Provers", *J. Assoc. for Comp. Mach.,* Vol. 33, October 1986, pp. 633-657.

9. Matt Kaufmann, "DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers", Tech. report 43, Computational Logic, Inc., May 1989.

10. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., MIT, *LISP 1.5 Programmer's Manual,* 1962.

11. J. R. Shoenfield, *Mathematical Logic,* Addison-Wesley, Reading, Ma., 1967.

12. Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J Strother Moore, "Functional Instantiation in First Order Logic", Tech. report 44, Computational Logic, Inc., May 1989.

13. Jussi Ketonen, "EKL - Ramsey Theorem", Tech. report, Department of Computer Science, Stanford University, December 1986.

14. Kenneth Kunen, *Set Theory: An Introduction to Independence Proofs,* North-Holland, New York, 1980.